Evaluating Design: A Formative Evaluation of Agent Development Environments Used For Teaching Rule-Based Programming

Mark A. Cohen
mcohen@lhup.edu
Business Admin, Computer Science, and Information Technology
Lock Haven University
Lock Haven, PA 17745, USA

Frank E. Ritter
ritter@ist.psu.edu

Steven R. Haynes
shaynes@ist.psu.edu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA

Abstract

We present two development environments designed to make it easier for students to create intelligent agents by taking advantage of established software engineering principles. This paper reports the results of a formative evaluation of the Herbal and the Vacuum Cleaner Environments. Findings from the study suggest design changes geared towards making these environments more useful for teaching rule-based programming and agent development.

Keywords: formative usability study, Herbal, intelligent agents, rule-based programming

1. INTRODUCTION

Teaching students how to program intelligent agents can be difficult. One way to simplify the task of teaching intelligent agent development is to improve the development tools by taking advantage of established software engineering principles such as high-level languages, maintenance-oriented development environments, and software reuse. These principles have recently been realized in the Herbal integrated development environment (Cohen, Ritter, & Haynes, 2005), which is a collection of tools that allows students and professional modelers to learn or engage in intelligent agent development by exploiting modern software engineering principles.

The Vacuum Cleaner Environment (Cohen, 2005) is another tool that can be used to help students learn agent development. This environment is simple enough to introduce to undergraduates, yet complex enough to allow for the creation of interesting agents. In addition, the environment is colorful and entertaining, thus holding the interest of students.

This paper reports the results of a formative evaluation (Scriven, 1967; Rosson & Carroll, 2002) of Herbal and the Vacuum Cleaner Environment, which lead to design changes

that have made both environments more useful for teaching agent development.

Overview of the Task

The Vacuum Cleaner Environment is based on a very simple virtual world introduced in a widely used Artificial Intelligence textbook, *Artificial intelligence: A modern approach* by Stuart Russell and Peter Norvig (2003). In the Vacuum Cleaner World, a vacuum cleaner resides in an environment that contains two squares: A and B. Each square can be either clean or dirty. The vacuum cleaner's percepts allow it to detect what square it is in and the state of the square (i.e., clean or dirty). In addition, the vacuum cleaner can perform four actions: move left, move right, clean, or do nothing. This environment is useful because its entire state space, consisting of only eight states, can be easily illustrated and explored, yet is complex enough to let us discuss efficiency and strategies. In addition, if a performance measure is used, the concept of agent rationality (Russell & Norvig, 2003) can be introduced.

There are several implementations of the Vacuum Cleaner World available. For example, the Pyro robotics toolkit (Blank, Kumar, Meeden, & Yanco, 2006) includes an implementation in Python. Another interesting extension of the Vacuum Cleaner World, created by Musicant and Exley (2004), allows students to program a physical robot to navigate a simplified version of the Vacuum Cleaner World. Additional implementations, in a variety of languages, are included on the official website for Artificial Intelligence: A Modern Approach (aima.cs.berkeley.edu).

While these implementations are useful for introducing basic agent programming concepts, they are either too simplistic for more advanced rule-based programming, or require the overhead of expensive hardware. To effectively evaluate Herbal, a custom graphical vacuum cleaner environment was created in Java (Cohen, 2005). This environment supports rule-based programs written in two widely used agent architectures: Jess (jessrules.com) and Soar (sitemaker.umich. edu). A screenshot of the Vacuum Cleaner Environment is shown in Figure 1.

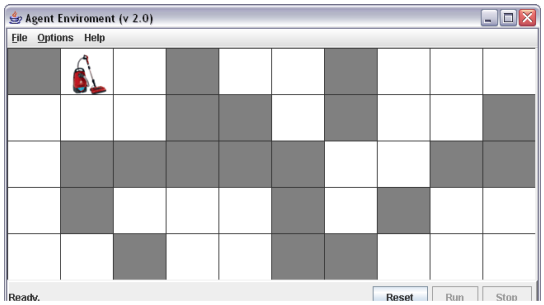

Figure 1: The Vacuum Cleaner Environment.

2. METHOD

This section describes the method used for the study conducted in parallel with an undergraduate artificial intelligence class. The goal of this study was to improve the design of Herbal and the Vacuum Cleaner Environment. Specifically, this study was designed to measure four different factors:

- The students' impressions of rule-based programming in general, and Jess specifically.
- The students' impressions of graphical development environments in general, and Herbal and the Vacuum Cleaner Environment specifically.
- The students' impressions of higher-level methods for organizing rules in general, and the use of the Problem Space Computational Model (PSCM) specifically. In the PSCM, behavior is defined as movement through a problem space, which is a high-level tool useful for partitioning knowledge (Newell, 1990).
- The students' impressions of the Herbal high-level language.

This study took advantage of cognitive dimensions research (Blackwell & Green, 2003) to evaluate the Herbal Integrated Development Environment. These dimensions provide a framework and a common vocabulary that can be used to judge the design of a notational system like Herbal. In addition, Blackwell and Green (2000) have shown that the use of cognitive dimensions in questionnaires can be useful for evaluating usability.

Table 1 shows the eight cognitive dimensions selected as usability evaluation criteria. These dimensions were chosen because they measure the degree in which the principles that mediated the design of Herbal were achieved (i.e., embracing high-

level languages, enabling reuse, and supporting maintenance-oriented development).

Table 1: The Cognitive dimensions used to evaluate the design of Herbal.

Cognitive Dimension	Description
Closeness of mapping	How closely does the behavior representation language match the way that the modeler describes the behavior?
Error-proneness	How easy is it to make errors using the behavior representation language?
Hidden dependencies	How easy does the behavior representation language make it to create hidden dependencies between model entities?
Premature commitment	How often is the developer forced to make a commitment in the model before there is enough information to make the commitment?
Provisionality	How easy is it to make provisional commitments that can be corrected at a later time? Provisionality allows modelers to easily examine design options and construct what-if scenarios.
Role-expressiveness	How easy is it to discover why a modeler has chosen a particular design? Explicit support for design rationale, as discussed earlier, improves a systems role-expressiveness.
Viscosity	How easy is it to make changes to an existing model? The less the viscosity, the easier it is to change the model.
Visibility	How easy is it to view the elements in a model, including their internal details?

Participants

The seven participants recruited for this study were undergraduate students majoring in Computer Science (CS) or Computer Information Science (CIS) at Lock Haven University; they were enrolled in an upper-level Artificial Intelligence course at Lock Haven. Enrollment in this course was the only requirement for participating in the study. Participants were not paid for taking part in this study. Seven students in the class agreed to participate: one CIS student and six CS students.

Apparatus

Participants used Dell Desktop computers running Linux to complete the required tasks. These desktops are all located in the Lock Haven Penguin Lab and are equipped with a keyboard, a mouse, a 100MB external hard-drive, and a 17-inch flat screen monitor.

The required software for this experiment was installed on each machine. The software was Eclipse (3.2.1), Java (1.5), Herbal (2.0.2 Pre-release D), Jess (6.1), the Vim text editor, and the Vacuum Cleaner Environment (2.0).

Design

As part of the course requirements, all students were asked to complete four assignments. The first assignment asked the participants to create a Jess program that simulated customers entering a bank and waiting in a queue for service. This assignment measured the participants' initial impressions of rule-based programming in Jess, and of graphical development environments in general.

The second assignment required the participants to create two vacuum cleaner models. The purpose of this assignment was to measure participants' impressions of rule-based programming in Jess, graphical development environments, and the Vacuum Cleaner Environment.

The third assignment asked the students to use Jess modules to create a vacuum cleaner agent that operated in the PSCM. The purpose of this assignment was to measure the participants' impressions of problem spaces and the PSCM from the perspective of organizing and modularizing rules.

The fourth assignment was to repeat assignment Three, but to use an early prototype of the Herbal high-level language and development environment to create the agent. The purpose of this assignment was to measure the participants' impressions of Herbal.

Data collection consisted of participant observation and quantitative and qualitative survey questionnaires derived from cognitive dimensions research (Blackwell & Green, 2000). Participant observations and open-ended survey questions were coded based on the cognitive dimensions in Table 1. Portions of the assignments were completed during class time so that participant observation could be conducted. Upon completion of each assignment, surveys were administered to the participants. Table 2 provides a summary of the four tasks performed by the participants.

Table 2: Summary of the experimental design for the formative evaluation.

Task	Data Collected	Purpose
Experiment 1: A Jess program modeling customers waiting in a queue at a bank	• The Jess source code • Completed survey • Participant observations	To measure student impressions of rule-based programming and graphical development environments
Experiment 2: A simple vacuum cleaner agent that cleaned a room A vacuum like the first one, but also keeps track of how many squares it cleaned	• The Jess source code • Completed survey • Participant observations	To see if the participants' impressions of rule-based programming and graphical development environments changed after using the Vacuum Cleaner Environment To measure the students' impressions of the Vacuum Cleaner Environment
Experiment 3: A vacuum that uses Jess modules and problem spaces	• The Jess source code • Completed survey • Participant observations	To measure the participants' impressions of problem spaces and the Problem Space Computational Model
Experiment 4: A vacuum that operated in problem spaces	• The Jess source code • Completed survey • Participant observations	To measure the participants' impressions of Herbal

Procedure

The study began with each participant reading and signing the consent form as well as completing a User Background Survey, which collected basic information about his or her background and expectations prior to participating in the study.

During the semester, participants were assigned each of the four assignments in order. When participants were given class time to work on the assignments, observations about the participant's performance, as well as the interactions between the experimenter and the participant, were noted by the experimenter. When participants finished each assignment, they were asked to complete a user reaction survey. The surveys were designed to measure the four objectives given in the Methods section.

The first assignment asked participants to create a Jess program that simulated customers entering a bank and waiting in a queue for service. The simulation operates by generating random numbers that determine how much time will elapse before the next customer enters the bank, and how much time it will take for the teller to service the current customer. The simulation was run for 1,000 simulated minutes, and during this time customers were added to a queue when they enter the bank and, as the teller becomes available, customers were removed from the queue to be serviced by the teller.

Participants worked alone on this assignment and used the Vim text editor to create their programs. Although difficult to control, participants were asked not to use graphical development environments and debuggers. When the assignment was finished, participants were each asked to complete User Reaction Survey #1.

The second assignment required the participants to create two vacuum cleaner agents. The first agent was a simple agent that cleaned a dirty room. This agent was run without the ability to remember facts (no state), no penalty for movement, no radar sensor, and in an environment two squares wide and one square tall. Participants were asked to record the best possible score for a run of 10 steps and the average score of their agent. The second agent operated in the same environment; however, this agent was allowed to maintain state and was assigned a penalty for each movement. Students were asked to minimize the penalty by remembering where the vacuum had been so it stopped moving when all squares were visited. Participants worked alone on this assignment and used the Vim text editor to create their programs. Again, graphical development environments and debuggers were discouraged. When the assignment was finished, participants were each asked to complete User Reaction Survey #2.

Problem spaces are simulated in Jess using Jess modules (Friedman-Hill, 2003). The third assignment asked the students to use Jess modules to create a vacuum cleaner agent that operated in problem spaces. The problem space hierarchy and the relationships between them are shown in Figure 2.

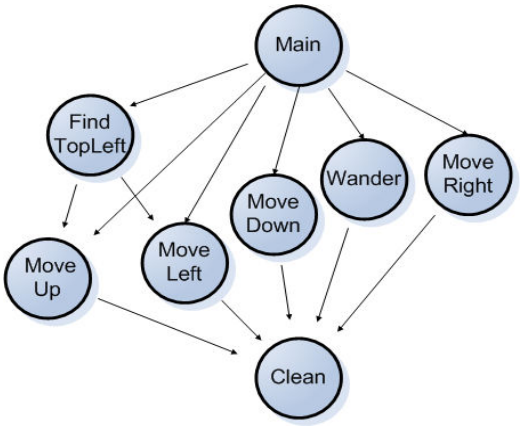

Figure 2: Problem space hierarchy for assignments 3 and 4.

When the agent in the third assignment started, it entered the *FindTopLeft* problem space, which caused it to go immediately to the top left square on the board, cleaning dirty squares along the way. The *FindTopLeft* problem space used the *MoveUp* and *MoveLeft* problem spaces to accomplish its goal and the *MoveUp* and *MoveLeft* problem spaces used the *Clean* problem space to make sure squares were cleaned along the way.

After the agent arrived at the top left square, it walked the perimeter of the board, cleaning any dirty squares it encountered during its travels. While the agent walked the perimeter, it was asked to assert the following three facts: a fact that represents the height of the board, a fact that represents the width of the board, a fact that represents the total number of squares on the board. The *MoveUp*, *MoveLeft*, *MoveDown*, and *MoveRight* problem spaces accomplished this behavior.

After the agent walked the entire perimeter, it entered a problem space called Wander that caused the agent to explore the board using the following algorithm. If the agent was on a dirty square, it cleaned it. If there was a dirty square adjacent to the agent, it should move to that square. If there were no dirty squares near the agent, it should randomly move to a new square, if the agent had visited every square on the board since it began to wander, it should stop moving.

As in the first two assignments, participants worked alone on assignment three and used the Vim text editor to create their programs. Graphical development environments and debuggers were forbidden. When the assignment was finished, participants were asked to complete User Reaction Survey #3.

The fourth assignment was to repeat assignment number three, but to use Herbal to create the agent, instead of Vim. Participants worked alone on assignment four. When the assignment was finished, participants were asked to complete User Reaction Survey #4.

3. RESULTS

Data were collected using surveys and participant observation. Many of the questions in the surveys were designed to measure the cognitive dimensions listed in Table 1. Although all of the participants completed each of the four required assignments, not all participants choose to

complete each survey (despite several reminders).

Table 4, Table 5, Table 6, and Table 7 (in the Appendix) show quantitative results for each of the four surveys. The number of participants that completed each survey is indicated in the caption of each table. In addition, if a question or result mapped to a cognitive dimension, it is indicated in the table.

Table 8 (in the Appendix) shows the qualitative results from Survey #4, and Table 9 (in the Appendix) shows the observations made while the participants were working on the assignments. The responses to the open-ended questions, and the observations made while programming, were coded based on the related cognitive dimensions. This coding is displayed in Table 8 and Table 9.

4. DISCUSSION

The small number of students enrolled in the Artificial Intelligence class limited the number of participants in this study to seven. This small sample size does make it difficult to generalize the study's findings. However, the following discussion suggests design changes that may help make the Herbal and Vacuum cleaner environments more useful for teaching rule-based programming and agent development.

Responses to the first two surveys (Table 4 and Table 5) indicate that after the first assignment, participants were divided about their comfort level with Jess syntax. Two out of six found the syntax challenging, one was neutral, and three did not find the syntax difficult at all. The level of comfort with Jess syntax was not surprising: especially because this evaluation was conducted in an upper-level, CS/CIS course using students with considerable programming experience.

The participants comfort level with Jess syntax increased after completing the second assignment, with five out of seven disagreeing with the statement that Jess syntax is difficult. Reasons for becoming more comfortable with Jess syntax could be related to gaining more experience with the language.

In addition, participants agreed that being able to view a running agent visually in a graphical environment would help make agent programming easier. They also expressed the need for more than just console output for debugging their agents. Responses to these same questions remained strong after they were introduced to the Vacuum Cleaner Environment in the second assignment.

Survey #2 (Table 5) shows participants were positive about the effectiveness of the Vacuum Cleaner Environment. Participants found that the environment made the programming assignments easier and more enjoyable. In addition, participants felt that the Vacuum Cleaner Environment was created with just the right amount of complexity.

Responses from Survey #3 (Table 6) validated the use of the PSCM as the foundation for the Herbal high-level language. Participants agreed that the PSCM made agent programming easier because it componentized their agents. In addition, responses showed that participants favored the idea of a development environment and debugger that supported the PSCM. Results from Survey #3 illustrate that a higher-level language that allows programmers to organize rules into higher-level structures was appreciated, and that the PSCM is a good choice for this purpose.

Results from Survey #4 (Table 7), which are directly related to the design of Herbal, are mixed. Most participants felt that they would rather program using pure Jess than the Herbal Development Environment. They also felt strongly that Herbal needed better visualizations of the agent's structure. In addition, participants were not convinced that Herbal made it easier to make changes to agent code. They also felt that Herbal forced them to work in a particular order when developing agents. This means that Herbal poorly supports the Visibility, Viscosity, Provisionality, and Premature Commitment dimensions. In addition, mixed responses from participants about the time it takes to learn and use Herbal also indicated a need for design changes.

However, some responses in Survey #4 were positive. For example, participants found it easier to reuse model components using Herbal than when programming using pure Jess. In addition, participants found

the XML high-level language used by Herbal to be easy to read and understand.

Interestingly, in Survey #4 half of the participants preferred programming by editing the Herbal XML high-level language, while the other half preferred the GUI editor. Herbal was designed to support both methods of programming because it was believed that preferences, and requirements, for both styles of programming exist (Powers, Ecott, & Hirshfield, 2007). These results support this design choice.

Responses to the open-ended questions (Table 8) and the participant observations (Table 9) were used to help discover the reasons behind some of the negative responses in Survey #4. These reasons were used to help improve the design of Herbal. For example, the frustration with the order that Herbal enforced while creating agents is evident in both the open-ended questions and the participant observations. Participants did not like having to provide a complete specification for a component at the time it was created. They also did not like having to remove references to a component before the component could be deleted. These problems made it difficult to create and change an agent. This feedback suggests the need for design changes for better support of the Viscosity, Provisionality, and Premature Commitment cognitive dimensions.

Another problem indicated in both the open-ended questions and participant observations was poor support for the Visibility cognitive dimension. Specifically, participants requested better visualizations of the model structure. The need for this type of visualization was also evident during participant observation.

Participants also had trouble getting comfortable with some of the terminology used by Herbal. For example, participants struggled with the difference between a problem space and an operator. Discussions with participants suggested that it helped to refer to problem spaces as behaviors or goals.

Participant observations and survey responses indicated that participants had trouble finding and fixing errors in their agents. One possible factor was that the console method of debugging caused execution to be traced using rules instead of the PSCM terminology used when the agent was created. Participants were trying to see what problem space their agent was in, and which operator was recently applied, but the trace they were using contained a list of rules. This mismatch between the behavior representation language and the way that trace describes the model's behavior resulted in poor support for the Closeness of Mapping dimension. A good debugger or tracing tool that maps directly to the PSCM rather than the rules would help here.

Finally, bugs within the Herbal GUI editor, that allowed participants to make fatal mistakes (e.g., the GUI editor stopped functioning when an invalid model configuration was accidentally created) that could only be fixed in the XML code, frustrated participants, and this frustration was evident during participant observation and in open-ended responses.

5. CONCLUSIONS

There were several important lessons learned during the formative study described here, and many of these lessons resulted in changes in the Herbal design. For example, participants felt strongly that Herbal needed a better visualization of the agent structure. This feedback resulted in the development of the Model Browser View in Herbal. This window shows a hierarchical view of a model's structure, giving the programmer a high-level picture of the model and its components.

In addition, participants were annoyed by the fact that Herbal forced them to work in a particular order when developing agents. To correct this problem, "soft" warnings were implemented in Herbal. During normal development, an agent is often only partially completed in a work session. With the addition of soft warnings, an incomplete agent produces a message that is passively displayed in the Eclipse output window. When a warning is displayed, the developer is allowed to continue without interruption. This makes it possible for developers to work in any order by building or editing models that are not yet complete.

The participants' difficulty debugging models indicated poor support by Herbal for Role Expressiveness and Closeness of Mapping. To correct this problem, working sets (Ko,

Aung, & Myers, 2005) that leveraged existing design rationale, were added to Herbal. These working sets should make it easier for modelers to find task relevant model components during maintenance. In addition, a graphical debugger was built that traces model execution using PSCM components rather than rules.

To correct the participants' problems with terminology some of the model components were renamed. For example, the concept of a behavior was introduced to help users understand problem spaces, and a Design Pattern Wizard was created to make it easy for users to create new model behaviors that are ultimately represented as problem spaces.

Finally, several bugs in the GUI Editor were discovered during this study. These bugs frustrated participants and made it difficult for them to complete the tasks. All the bugs identified during the formative evaluation were fixed.

Results from this study also helped confirm many of the design decisions that were made early on in development process. For example, the choice to use the PSCM as the basis of the Herbal high-level language was confirmed by participants, as they indicated that the PSCM made agent programming easier. In addition, the emphasis on reuse during Herbal's design was successful as participants found it easier to reuse model components using Herbal.

The decision to use XML for Herbal's high-level language was also supported by this study. Finally, the design decision to allow users to edit Herbal code using both the GUI Editor and by directly editing the XML code was appreciated by participants. Table 3 summarizes the lessons learned during this study, and the changes that were implemented to address these lessons.

Table 3: Summary of the design changes resulting from the formative study.

Formative Result	Design Change
Herbal needed a better visualization of the agent structure	Added a Model Browser View
Herbal forced them to work in a particular order	Implementation of "soft" warnings
Difficulty debugging models	Implementation of working set feature that leverages existing design rationale, and a graphical debugger based on the PSCM
Problems with some PSCM terminology	Aliases renamed to input/output variables, impasses presented as conditions for entry, and behavior design pattern associates problem spaces with agent behaviors
Participants encountered frustrating bugs in the GUI editor	Bugs fixed

6. REFERENCES

Blackwell, A. F., & Green, T. (2000). "A cognitive dimensions questionnaire optimised for users." Proceedings of the 12th Annual Meeting of the Psychology of Programming Interest Group, 137-152.

Blackwell, A. F., & Green, T. (2003). Notational systems: The coginitive dimensions of notations frameworks. In J. M. Carroll, HCI models, theories, and frameworks (pp. 103-133). San Francisco, CA: Morgan Kaufmann.

Blank, D. S., Kumar, D., Meeden, L., & Yanco, H. (2006). "The Pyro toolkit for AI and robotics." AI Magazine , 27.

Cohen, M. A. (2005). "Teaching agent programming using custom environments and Jess." The Newsletter of the Society for the Study of Artificial Intelligence and the Simulation of Behavior , 120, p. 4.

Cohen, M. A., Ritter, F. E., & Haynes, S. R. (2005). "Herbal: A high-level language and development environment for

developing cognitive models in Soar." Proceedings of the 14th Conference of Behavior Representation in Modeling and Simulation (pp. 133-140). Orlando, FL: U. of Central Florida.

Friedman-Hill, E. (2003). Jess in action: Rule-based systems in Java. Greenwich, CT: Manning Publications Company.

Ko, A. J., Aung, H. H., & Myers, B. A. (2005). "Eliciting design requirements for maintenance-oriented IDEs: A detailed study of the corrective and perfective maintenance tasks." Proceedings of the International Conference on Software Engineering (pp. 126-135). New York: ACM Press.

Musicant, D. R., & Exley, A. (2004). "Easy integration of LEGO Mindstorms into vacuum world simulations." Proceedings of the Special Interest Group on Computer Science Education. Norfolk, VA: ACM Press.

Newell, A. (1990). Unified theories of cognition. Cambridge, MA: Harvard University Press.

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). "Through the looking glass: teaching CS0 with Alice." Proceedings of the Special Interest Group on Computer Science Education, 39 (1), 213-217.

Rosson, M. B., & Carroll, J. M. (2002). Usability engineering: Scenario-based development of human-computer interaction. San Francisco, CA: Morgan Kaufmann.

Russell, S., & Norvig, P. (2003). Artificial Intelligence: A modern approach. Supper Saddle River, NJ: Prentice Hall.

Scriven, M. (1967). The methodology of evaluation. In R. Tyler, R. Gagne & M. Scriven (Eds.), Perspectives of curriculum evaluation (pp. 39-83). Chicago: Rand McNally.

Appendix

Table 4: Quantitative results from User Reaction Survey #1 (N=6).

Impressions of rule-based programming and graphical development environments

I understand the main constructs in Jess but I find it difficult to implement them because the Jess syntax is difficult.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**2**	**1**	**3**	0

Programming agents would be easier if the behavior of my running agent was displayed visually in a graphical environment.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**4**	**1**	0	0

Using print statements to print the progress of my agent in a console window is all want in order to help me create and debug my agents.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	0	**3**	**3**	0

I would enjoy programming in Jess more if there were a better development environment.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**2**	**3**	0	0

Table 5: Quantitative results from User Reaction Survey #2 (N=7).

Impressions of rule-based programming, graphical development environments, and the Vacuum Cleaner Environment

I understand the main constructs in Jess but I find it difficult to implement them because Jess syntax is difficult.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**1**	**1**	**5**	0

Programming agents would be easier if the behavior of my running agent was displayed visually in a graphical environment.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
2	**4**	**1**	0	0

Using print statements to print the progress of my agent in a console window is all want in order to help me create and debug my agents.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	0	**2**	**5**	0

The vacuum cleaner graphical agent environment made programming agents more fun.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
4	**3**	0	0	0

The vacuum cleaner graphical agent environment made it easier to learn how to create rule-based agents.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**4**	**2**		

The vacuum cleaner graphical agent environment had just the right amount of complexity to make it possible to create interesting agents without getting distracted by the details of the environment.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**6**	0	0	0

Table 6: Quantitative results from User Reaction Survey #3 (N=6).

Impressions of problem spaces and the Problem Space Computational Model

The ability to group a set of operators and behavior into a problem space makes it easier to create complicated agents.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**4**	**2**	0	0

A graphical environment that simplified the use of problem spaces, operators, and impasses is needed to make them useful in Jess.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**1**	**3**	**1**	0

Breaking my agent code into problem spaces made it possible to breakup complicated agent behavior into smaller, less complicated parts.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
3	**2**	**1**		

It would be easier to use problem spaces if there was a graphical debugger that showed my agent as it moved from problem space to problem space.

Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**3**	**2**	0	0

Table 7: Quantitative results from User Reaction Survey #4 (N=4).

Impressions of the Herbal Prototype				
If given the choice, I would rather use Herbal that pure Jess in order to complete the agent programming assignments given in this course.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	0	0	**2**	**1**
Herbal would be easier to use if there were better visualizations of the agent structure.			*Measures Visibility*	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
3	**1**	0	0	0
It takes less time to create an agent using Herbal that to write code in Jess.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**1**	**1**	**1**	0
It takes less time to learn how to use Herbal than to learn how to write Jess Code.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**2**	**1**	**1**	0
The Herbal GUI editor makes it easier than Jess programming to recognize components of my agent (problem spaces, operators, etc.).			*Measures Visibility*	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**1**	**2**	**1**	0
Herbal makes it easier than Jess to reuse conditions and actions in my agent.				
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	**1**	**2**	0	0
Herbal's XML language is easy to read/understand.			*Measures Closeness of Mapping*	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**2**	**2**	0	0
I would rather write code in Herbal using thee XML high-level language than with the GUI editor.			*Measures Closeness of Mapping and Viscosity*	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**2**	0	**2**	0
Herbal makes it easier than Jess to change my agent.			*Measures Viscosity*	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1	0	0	**3**	0
Herbal placed very little restrictions on the order in which I created my agent.			*Measures Provisionality and Premature Commitment*	
Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
0	**1**	**1**	**2**	0

Table 8: Qualitative results from User Reaction Survey #4.

Impressions of the Herbal Prototype

What part of Herbal did you find most useful?

Response	# Responding	Cognitive Dimension
Syntax becomes a non-issue	2	Closeness of mapping
Wiring aliases	1	N/A

What part of herbal did you find most confusing?

Response	# Responding	Cognitive Dimension
Understanding the order in which to create components	2	Provisionality and Premature Commitment
Wiring aliases	2	N/A
Getting a high-level picture of the agent structure	1	Visibility

If you were in charge of programming Herbal, what improvements would you make?

Response	# Responding	Cognitive Dimension
Visual representation of the model structure	3	Visibility
Wizard or flow-chart that helps you create components	2	Provisionality and Premature Commitment

Table 9: Observation of participants completing assignment 4.

Observation	Cognitive Dimension
Participants had problems understanding what an alias is in Herbal. They struggled with this term. Discussions with participants revealed that it helped them to think of them as input and output variables.	Closeness of Mapping
Participants had problems understanding when you would want to use a problem space as opposed to just an operator. Thinking of the problem space as a behavior seemed to be very helpful.	Closeness of Mapping
Participants had a hard time understanding the term impasse. It helped to explain the impasse as a set of conditions that cause entry into a problem space.	Closeness of Mapping
Participants had problems debugging common problems. For example, they struggled figuring out why an agent was not entering a specific problem space or why an operator was not firing.	Role-expressiveness Hidden Dependencies
Participants were frustrated by the requirement to fully specify a component when it was created.	Provisionality, Premature Commitment Viscosity Hidden Dependencies
Participants were frustrated when the system forced them to delete all references to a component before they could delete the component.	Provisionality, Premature Commitment Viscosity
Participants were frustrated by the lack of warnings. The system produced errors for situations that occur during development but were easily corrected later in the development process. There errors were highly dependent on the order in which the model was created. The participants would prefer these to be reported as warnings.	Provisionality, Premature Commitment Viscosity
In some cases, participants were allowed to make certain mistakes that caused the visual editor to stop functioning and could only be fixed using the XML code.	Error-proneness
Participants continued to express the need for a high-level visualization of the model and its structure.	Visibility
Participants continually commented that they would have rather learned Herbal and then Jess instead of the other way around. They all felt that Herbal is useful in learning how to program in pure Jess.	N/A